テンプレートメソッドの形成に基づく類似メソッド集約支援

Support for Merging Similar Methods Based on Form Template Method

政井 智雄*　吉田 則裕†　松下 誠‡　井上 克郎§

あらまし　ソフトウェア保守のコストを増大させる要因として，コードクローンが指摘されている．類似メソッド対（コードクローンを共有するメソッドの対）間のコードクローンは，“テンプレートメソッドの形成”リファクタリングを行うことで集約できる．本研究では，このような集約作業を支援する手法を提案する．

1　まえがき

ソフトウェア保守を困難にする要因の 1 つとしてコードクローン [1–3] が指摘されている．コードクローンとは，ソースコード中に含まれる一致もしくは類似したコード片を持つコード片のことである．類似メソッド対（コードクローンを共有するメソッドの対）を集約する方法の 1 つとして，“テンプレートメソッドの形成”リファクタリング [4] が挙げられる．このリファクタリングを行う開発者は，(1) 類似メソッド対を各メソッドの固有処理と共通処理に分割し，(2) 共通処理を親クラスのメソッドに実装し，(3) 固有処理を各子クラスのメソッドに委譲するように書き換える．しかし，これらを行うためには，類似メソッド対間の差分を特定し，類似メソッド対間の差分が共通化されるようにメソッド抽出を行う必要がある．この作業は，各メソッドを対比させながら行う必要があり，人手では時間がかかると考えられる．

本研究では，これら作業を支援するために，固有処理として切りだすコード片の候補を提示する手法を提案する．本手法は，類似メソッド対間の構文上の差分を検出し，その後差分を含みかつメソッド抽出に適したコード片を固有処理として切りだす候補として検出する（図 4 にツールの出力掲載）．候補の検出では，まず差分と同一の範囲に対してメソッド抽出に適しているか判定し，その後段階的に範囲を拡大させながら，同様の判定を行う．

2　コードクローンを集約するリファクタリング

コードクローン間の差異に基づいた分類を Bellon は定義している [5]．その 1 つであるタイプ 3 のコードクローンは，あるコード片をコピーアンドペーストした後，開発者が文の挿入や削除を行うことで作成される（図 1 参照）．以降，タイプ 3 のコードクローンを，**差分を含むコードクローン**と呼ぶ．

リファクタリングとは“ 外部から見たときの振る舞いを保ちつつ，理解や修正が簡単になるように，ソフトウェアの内部構造を変化させること ”[4] である．差分を含むコードクローンに効果的であると言われているリファクタリングパターンの 1 つが“ テンプレートメソッドの形成 ”リファクタリング [4] である．

“ テンプレートメソッドの形成 ”リファクタリングは，以下の 3 つのステップに分けられる．

*Tomoo Masai, 大阪大学

†Norihiro Yoshida, 奈良先端科学技術大学院大学

‡Makoto Matsushita, 大阪大学

§Katsuro Inoue, 大阪大学

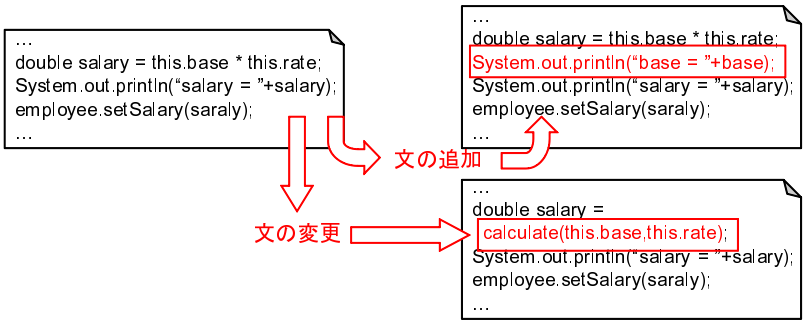

図 1　差分を含むコードクローンの例

ステップ A　差分を固有の処理として切り出すためにメソッドとして抽出する差分を含むコード片を決定する．
ステップ B　固有の処理を行うメソッドとしてコード片を抽出する．
ステップ C　共通の処理となった類似メソッド対を引き上げる．

上記の 3 つのステップで“ テンプレートメソッドの形成 ”リファクタリングが完了し，コードクローンを集約することができる．このリファクタリングによって，2 つの抽象メソッドを実装するだけで子クラスを追加できる拡張性の高い設計となり，また類似メソッドが増加する可能性も低くなる．しかし，これらのステップのうちステップ A，B において，以下の問題が発生する場合がある．

ステップ A の問題　メソッドとして抽出する場合に，戻り値が複数必要であったり，制御文がまとまりを持っていないコード片はそのまま抽出することができない．
ステップ B の問題　メソッドとして抽出した結果，引数として渡す変数や，値を戻す変数が異なる場合がある．この場合，類似メソッド間の記述が一致せず，共通のメソッドとして親クラスへ引き上げることができない．

どちらのステップにおける問題も，抽出するコード片の範囲をそれぞれ拡大することで解決することができる．しかし，類似メソッド間で対応する位置関係にある各コード片の範囲を拡大し，逐一各コード片がメソッドとして抽出可能であるか，また抽出後に引数として渡す変数，値を戻す変数が一致しているかを判断する作業は，大きな労力を要する．

3　提案手法

本節では，提案手法の概要，また提案手法の実装について述べる．本手法は，統合開発環境 Eclipse の Java 開発キットのプラグインとして実装することで，Eclipse の既存の機能を利用し，かつ本手法を Eclipse を用いたコーディングの過程で利用出来るようにした．本手法は入力として類似メソッド対が与えられるとし，動作を以下の 4 つのステップに分け，ステップ 1〜3 において節 2 のステップ A の問題の解決，ステップ 4 において節 2 のステップ B の問題の解決をそれぞれ支援する．

3.1　[ステップ 1] 抽象構文木の生成

Eclipse の抽象構文木生成の機能を用いて，用意された類似メソッド対の抽象構文木をそれぞれ生成する．この抽象構文木を成すノードは，大きく以下の 2 つに分類できる．

タイプ A(値や子ノードを持つノード)　”ユーザ定義名”，”return 文”，”代入文”などのノードが該当する．”ユーザ定義名”のノードは値を持ち，”return 文”のノードは子ノードを持つ．”代入文”のノードは値，及び子ノードを持ってい

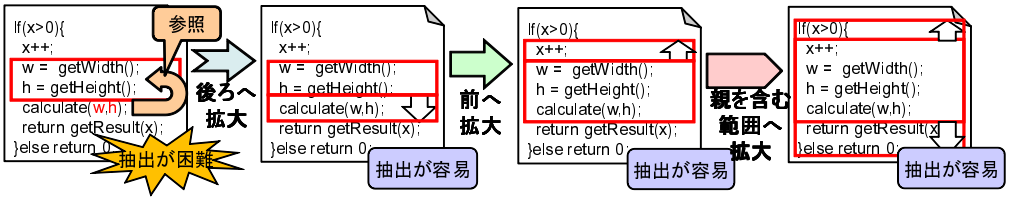

図 2　メソッド抽出が容易な部分木列検出の概要

る．”break 文”のノードのように，子ノードも値も持たない場合もある．

タイプ B(子ノードの列を持つノード)　{} で囲まれた記述が該当する．セミコロンで区切られるステートメントや，{} で区切られる記述がそれぞれ子ノードとなる．

3.2　[ステップ 2] 差分となっている部分木の検出

ステップ 1 で生成された 2 つの抽象構文木の比較を行い，抽象構文木間の差分となっている部分木を検出する．

抽象構文木の比較は，メソッド宣言に対応するノードから開始し，子ノードへと再帰的に比較を繰り返すことで行う．ノードの比較は以下の 2 つの操作から成る．

種類の比較　ノードの種類が同じか比較する．異なっていれば互いに対応関係のある差分として検出する．同じであれば，タイプによって以下の比較を行う．

値，子ノードの比較　タイプ A のノードの場合，ノードの持つ値や，子ノードをそれぞれ比較する，異なっていれば，互いに対応関係のある差分として検出する．

列の比較　タイプ B のノードの場合，子ノードの列を比較する．比較には，動的計画法を用いた既存の類似文字列マッチングアルゴリズム [6] を用いた．

差分と判断されたノードの子ノードは全て差分と判断するため，差分となるノードを以降，差分となる部分木と呼ぶ．このとき，差分となる部分木は，ソースコード上において，1 つのステートメントとなっている．

3.3　[ステップ 3] 抽出が容易な部分木の検出

差分となる部分木を含む，メソッドとして抽出可能な部分木，または部分木の列（タイプ B のノードの子ノード列の部分列）を検出する．

部分木が表すソースコード中のコード片が，メソッドとして抽出するコード片として適切か否かは，Eclipse の“ メソッドの抽出 ”リファクタリングを行う機能の事前条件判定機能を用いて判定する．この事前条件判定機能がコード片を，適切でないと判定する条件の中で本手法に関係するものは以下の 3 つである．

条件 1　複数の変数の初期化を含む宣言文，または複数の変数への代入文が含まれており，かつそれらの変数が後のコードにおいて参照されている．

条件 2　break 文，continue 文が含まれているが，それらに対応する制御文が含まれていない．

条件 3　戻り値を持たない return 文を含んでいる．

これらの条件を満たすコード片は，そのままメソッドとして抽出することができない，または抽出した場合に動作の不変を保証することができないコード片である．よってこのようなコード片は，メソッドとして抽出するコード片として検出せず，段階的に範囲を拡大し，判定を繰り返すことでこれらの条件を満たさない，抽出することが容易なコード片を表す部分木列の検出を行う．（図 2 参照）．

他の検出された差分となる部分木それぞれからも同様に拡大を行なう．また，既に抽出が容易として検出された部分木からも範囲の拡大を繰り返し行い，抽出範囲の全ての組み合わせが検出されるまでは検出を行う．

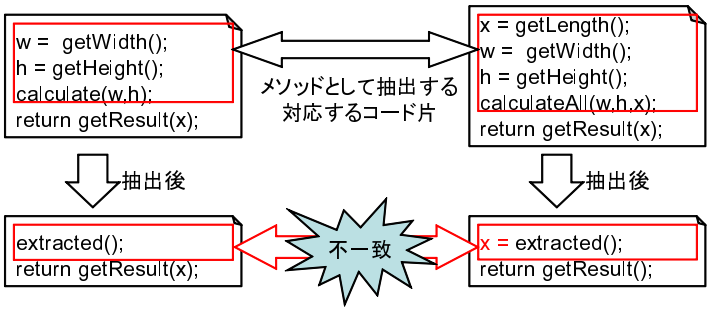

図 3　抽出後の記述が一致しない例

3.4　[ステップ 4] 部分木列の分類

検出されたメソッド抽出が容易な部分木列が表すコード片を，実際にメソッドとして抽出した後のメソッド呼び出し文の差異に基づいて分類する．このような差異が存在する場合，単純なメソッドの抽出だけではメソッド呼び出し文を一致させることができず，容易に親クラスへと引き上げることができない（図 3 参照）．よって，分類を行なった上でユーザに提示することで，リファクタリングが容易な候補の理解を促す．

3.5　手法の利用方法

Eclipse の Java 開発キットのプラグインとして実装した本手法の利用方法について述べる．本手法を利用するユーザは，以下の手順を行う．

1. リファクタリングの対象とする類似メソッド対を決定する．
2. Eclipse のメニューから，プラグインにより追加されたメニューを選択する．
3. ウィザードの指示に従い，類似メソッド対を選択する．
4. 図 4 のようにウィザードに表示される候補を確認し，リファクタリングを行う．

提示される候補は，分類ごとにタブアイテムに分けられ，タブアイテムを切り替えることでそれぞれ閲覧することが可能になっている．図 4 においては，全てのメソッド呼び出し文が一致する候補の 1 つを表示している．

4　適用実験

本節では，3 節で説明した提案手法を，実際のソースコードに適用した実験について述べる．提案手法は，リファクタリングを支援することを目的としている．そのため，提示される候補がリファクタリングに有効であり，またリファクタリングが容易となる候補が特定されていなければならない．よって本実験では，提示される候補の妥当性，及び候補に対して行う分類の妥当性をそれぞれ確認することを目的とした．

4.1　準備

適用実験では，Java，C#，C++等に対応したコンパイラ・コンパイラであるオープンソースソフトウェアの ANTLR2.7.4[1]内のメソッド対を対象に用いた．また，ANTLR2.7.4 から，提案手法を適用する類似メソッド対を検出するためのコードクローン検出ツールとして，Scorpio [7] を使用した．Scorpio はプログラム依存グラフを用いるコードクローン検出ツールである．プログラム依存グラフを用いたコードクローン検出は，差分を含むコードクローンを検出することができるという長所を持つ．Scorpio を用いて ANTLR2.7.4 から検出した差分を含むコードクロー

[1] http://www.antlr2.org/download/antlr-2.7.4.tar.gz

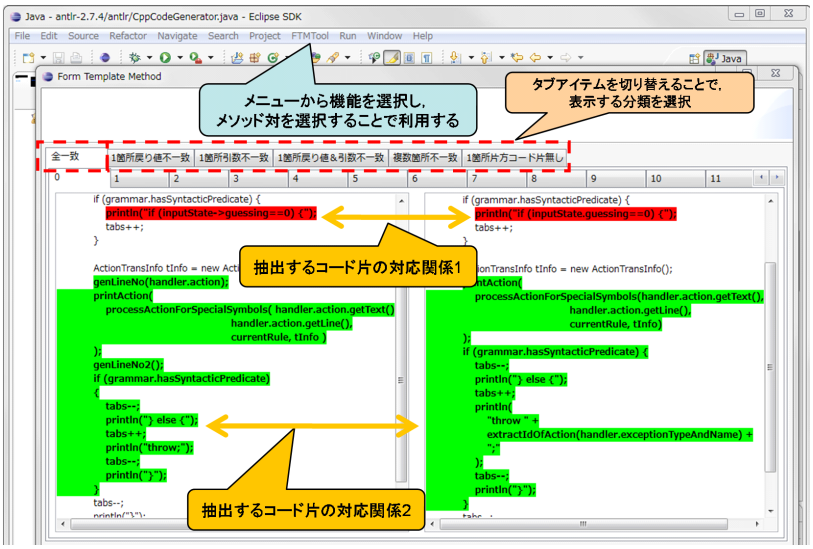

図 4　実装したプラグインを用いた候補の表示

ンのペアの中から，メソッド間クローン率が大きいコードクローンのペアの上位 3 組を特定し，それらをそれぞれ含むメソッドのペアを実験の対象に決定した．ここで，メソッド間クローン率 $R(m_1, m_2)$ は以下のように定義される．

$$R(m_1, m_2) = \frac{1}{2}\left(\frac{LOC_{c_1}}{LOC_{m_1}} + \frac{LOC_{c_2}}{LOC_{m_2}}\right)$$

ここで LOC_{m_1}，LOC_{m_2} はそれぞれ，“ メソッド m_1，m_2 の定義を含む行の数 ”とし，LOC_{c_1} は，“ m_1 の定義を含む行のうち m_2 の一部とのコードクローンを含む行の数 ”，LOC_{c_2} は，“ m_2 の定義を含む行のうち m_1 の一部とのコードクローンを含む行の数 ”とする．

節 3.4 で触れた条件 [8] を用いて，検出される候補を分類 α，分類 β に分類した．分類 α は，抽出後の全てのメソッド呼び出し文が一致する候補であり，分類 β は一致しないメソッド呼び出し文が存在する候補である．対象とした 5 組，及びそれぞれのクローン率 $R(m_1, m_2)$，それぞれから検出される候補の数を表 1 に示す．以降，順位 1〜3 に該当する類似メソッド対をそれぞれ，p_1, p_2, p_3 と呼ぶ．

4.2　提案手法を用いたリファクタリング

提示された候補の中から，分類ごとに 10 個ずつ無作為に選択し，実際にリファクタリングを行った．結果として，分類 α を用いたリファクタリングは，提示されたコード片をそのままメソッドとして抽出することで差異を取り除くことができ，差異を取り除くための操作（文献 [8] 参照）が必要であった分類 β を用いたリファク

表 1　対象としたメソッド

メソッド対	クローン率 $R(m_1, m_2)$	メソッド m_1	メソッド m_2	分類 α	分類 β	総数
p_1	0.878	CppCodeGenerator. genErrorHandler()	CSharpCodeGenerator. getErrorHandler()	32	282	314
p_2	0.728	CppCodeGenerator. genErrorTryForElement()	CSharpCodeGenerator. genErrorTryForElement()	9	0	9
p_3	0.575	CSharpCodeGenerator. setGrammarParameters()	JavaCodeGenerator. setGrammarParameters()	0	102,144	102,144

タリングに比べ，親クラスへと引き上げることが容易であることを確認した．
さらに，それぞれのリファクタリングを行ったソフトウェアに対し，86 個のテストケースを用いたテストを行い，リファクタリングの前後において外部的動作に変化がないことを確認した．

4.3 考察

候補の妥当性について 提示された候補を用いたリファクタリングの前後で，外部的動作に変化がなかったため，リファクタリングを支援する，抽出するコード片の候補としての妥当性を確認できたと考えられる．

候補の分類の妥当性について 分類 β に比べ分類 α の候補を用いたリファクタリングでは，必要な操作が少なくリファクタリングが容易であることを確認できたため，検出された候補に対し行った分類の妥当性を確認できたと考えられる．しかし p_3 においては，分類 α の候補が検出されなかった．よって，p_3 のような類似メソッド対に対しても，リファクタリングにおいて有効な候補を提示できるよう，新たな分類を考案することもひとつの課題と考えられる．

5 関連研究

Juillerat らは,“ テンプレートメソッドの形成 ”リファクタリングを行う作業を自動化する手法を提案している [9]．Juillerat らの手法は，本手法と同様にソースコードから生成した抽象構文木を用いて，差分となる部分木を検出した上で，メソッドの抽出を行うが，範囲を変化させず抽出後の形を 1 つとすることで目的とする自動化を行う．

6 まとめと今後の課題

本研究では，類似メソッド対に対し,“ テンプレートメソッドの形成 ”リファクタリングを行うための支援手法を提案した．今後の課題として，提示される候補数の削減，及び複数の類似メソッドの集約支援が考えられる．

謝辞 本研究は，日本学術振興会 科学研究費補助金 基盤研究（A）（課題番号:21240002），および基盤研究（C）（課題番号:22500026），研究活動スタート支援（課題番号:22800040）の助成を得た．

参考文献

[1] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. In *Proc. of ICSE 2007*, pp. 96–105, Minneapolis, MN ,USA, 2007.
[2] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng.*, Vol. 28, No. 7, pp. 654–670, 2002.
[3] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proc. of SAS 2001*, pp. 40–56, Paris, France, 2001.
[4] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison Wesley, 2000.
[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluationof Clone Detection Tools. *IEEE Trans. Softw. Eng.*, Vol. 33, No. 9, pp. 577–591, 2007.
[6] R. B. Yates and B. R. Neto. *Modern Information Retrieval.* Addison Wesley, 1999.
[7] 肥後芳樹, 楠本真二. 実規模ソフトウェアへの適用を目的としたプログラム依存グラフに基づくコードクローン検出法. ソフトウェアエンジニアリング最前線 2009, pp. 97–104, 2009.
[8] 政井智雄, 吉田則裕, 松下誠, 井上克郎. 類似メソッドの集約のための差分抽出支援. 電子情報通信学会技術研究報告 SS2010-8, Vol. 110, No. 60, pp. 45–50, 2010.
[9] N. Juillerat and B. Hirsbrunner. Toward an Implementation of the “Form Template Method ”Refactoring. In *Proc. of SCAM 2007*, pp. 81–90, Paris, France, 2007.